MTAT.03.227 Machine Learning (Spring 2012)
Exercise session V: Linear classification

Konstantin Tretyakov

March 13, 2012

The aim of this exercise session is to get acquainted with the basics of linear classification methods. In particular, we are going to explore the following topics:

1. Algebra and geometry of *linear functionals* and *linear transformations*.
2. Fisher's linear discriminant.
3. The least squares approach to classification.
4. The perceptron algorithm.

For that we shall go through 17 exercises, each worth 1 point and presumably doable in under 20 minutes. For each exercise you typically need to write a short piece of code and perhaps a couple sentences of your opinion about what you did and saw. You can submit your whole solution as a single R file with comments, provided it is formatted to be sequentially readable and executable. As usual, the maximum point count you may aim for is 10, but I'm sure doing all 17pts won't hurt.

I provide you some base code to build your solutions upon (`linear_class.R`).

Linear Functionals

The theory of linear classification (as well as linear regression) revolves around the concept of a *linear*[1] *functional*[2]. A linear functional is simply a function $f : \mathbb{R}^m \to \mathbb{R}$ of the form:

$$f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = w_1x_1 + w_2x_2 + \cdots + w_mx_m.$$

A linear functional is uniquely defined by a *weight vector* $\mathbf{w}$ and in the following set of exercises we shall try to gain some intuition as to how it feels and looks like.

[1] In mathematics, a *linear* function is a function that satisfies $f(\alpha x + y) = \alpha f(x) + f(y)$.

[2] In mathematics, a *functional* is a general term which refers to a function which inputs a vector and outputs a scalar value.

The function `plot.classifier`, shown below (also present in the base code), takes as input a two-dimensional weight vector `w` and visualizes the corresponding linear functional $f_{\mathbf{w}}$ as a filled contour plot[3]

```
plot.classifier = function(w) {
  x1s = seq(-10, 10, 0.2)
  x2s = seq(-10, 10, 0.2)
  fvals = outer(x1s, x2s,
            Vectorize(function(x1, x2) { x1*w[1] + x2*w[2] }))
  image(x1s, x2s, fvals,
        col=terrain.colors(40), breaks=-20:20, asp=1)
  contour(x1s, x2s, fvals, levels=-40:40, add=T)
  arrows(0, 0, w[1], w[2], length=0.2, lwd=4)
}
```

Exercise 1 (1pt). Familiarize yourself with the code. Apply it on the weight vector `w = c(1, 0.5)` and study the resulting plot.

Linear functional

1. By just looking at the plot, guess the length of the weight vector $\mathbf{w}$.
2. Verify your guess by computing the actual length of $\mathbf{w}$ (show the code for computing the length).
3. By just looking at the plot, guess two arbitrary points $\mathbf{x}$ which would have $f_{\mathbf{w}}(\mathbf{x}) \approx 3.5$. Add them to the plot.

   Hint: Use `points(x[1], x[2])` for adding a point `x` to the plot.
4. Verify your guesses by computing actual $f_{\mathbf{w}}(\mathbf{x})$ for the two points.
5. Guess how the picture will look like if you increase the length of $\mathbf{w}$ two-fold. What would be the values of $f_{\mathbf{w}}(\mathbf{x})$ for your selected points? Verify your guesses. Do you see that the reasoning you used to guess the length of $\mathbf{w}$ in step 1 might have been wrong?

As a side-note, this is a nice place to introduce you to R's *generic function* capabilities. Try this:

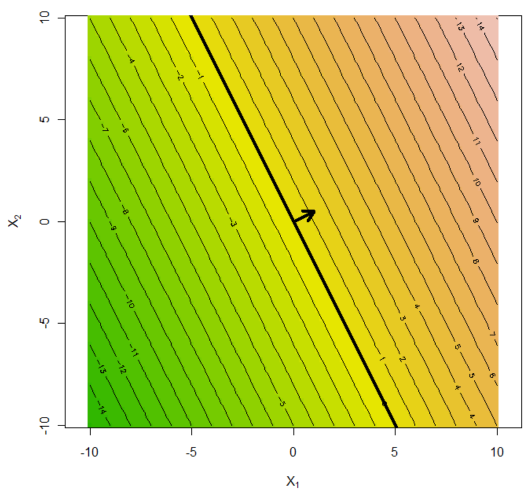
```
class(w) = "classifier"
plot(w)
```

Do you see how R automatically selects the `plot.<xxx>` function based on the `class` attribute of an object? If interested, read more using `help(class)`.

Exercise 2 (1pt). Modify the function `plot.classifier` to also draw a fat line, corresponding to all points where $f_{\mathbf{w}}(\mathbf{x}) = 0$. The result should look as follows:

Separating line

[3] Note, we do not use R's built-in function `filled.contour` because it is bad (it messes up the coordinate system of the plot).



Hint: `abline(?, ?, lwd=4)`.

Exercise 3-5 (3pt). Sometimes it is more convenient to regard a linear classifier as a function of the form[4]: *Bias term*

$$f_{(\mathbf{w},w_0)}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0.$$

Modify the function `plot.classifier` so that it would accept two parameters: $\mathbf{w}$ and w_0 (i.e. `plot.classifier = function(w, w0=0)`) and visualize the output of the corresponding function $f_{(\mathbf{w},w_0)}$. In particular, it should show the appropriate separating line using `abline`. You may keep the `arrows` invocation intact. Play with the visualization to get a feel at how w_0 affects the output.

1. Let `w = c(3, 4)`. Find `w0` such that the separating line passes through point `p = c(2.5, 0)`. What is the general formula?

   Hint: $f_{(\mathbf{w},w_0)}(\mathbf{p}) = 0$

2. Find `w0` such that the separating line is at distance 2.5 from zero. What is the general formula?

   Hint: $\mathbf{p} = \frac{2.5\mathbf{w}}{\|\mathbf{w}\|}$ is at distance 2.5 from zero.

3. By how much exactly does the separating line shift from 0 for a given value of w_0? What is the general formula?

   Hint: The ordering of these three questions is intentional.

[4]Formally, such a function is not a *linear functional* any more, this is an *affine functional*.

Linear Transformations

Whence a *linear functional* maps a vector to a number, a *linear transformation* maps a vector to a vector. A linear transformation of m-dimensional vectors is always an $m \times m$ matrix.

Exercise 6 (1pt). Generate and visualize a dataset of normally-distributed points:

Linear transformations

```
X = matrix(rnorm(100),ncol=2)
plot(X[,1],X[,2], asp=1)
```

1. Let $\mathbf{M}$ be a tranformation matrix. The application of this matrix to a point (i.e. column-vector) $\mathbf{x}$ can be computed using the expression $\mathbf{Mx}$. What is the correct expression to apply the transformation $\mathbf{M}$ to all rows of $\mathbf{X}$?

2. Define in R the following matrix:

$$\mathbf{R} = \begin{pmatrix} \cos(0.2) & -\sin(0.2) \\ \sin(0.2) & \cos(0.2) \end{pmatrix}.$$

   Apply it to all rows of dataset `X` to obtain a transformed dataset `Xt`. Add the transformed point to your plot as follows:

```
points(Xt[,1], Xt[,2], col='red')
segments(X[,1], X[,2], Xt[,1], Xt[,2], col='red')
```

3. Repeat the same for the following matrix:

$$\mathbf{S} = \begin{pmatrix} 3 & 0 \\ 0 & 0.5 \end{pmatrix}.$$

4. Finally, let $\mathbf{M} = \mathbf{RS}$. Try to guess, what $\mathbf{M}$ does to the points. Then verify your guess.

Exercise 7-8 (2pt). The normally-distributed data `X` that we generated in the previous exercise adheres to the following probability law:

Covariance normalization

$$\Pr[\mathbf{x}] = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\mathbf{x}^T\mathbf{x}\right)$$

Now that we transform the points $\mathbf{x} \to \mathbf{Mx}$, it holds that:

$$\Pr[\mathbf{Mx}] \propto \exp\left(-\frac{1}{2}\mathbf{x}^T\mathbf{x}\right).$$

Now denote $\mathbf{x}' = \mathbf{Mx}$ and substitute into the above equation to obtain something of the form:

$$\Pr[\mathbf{x}'] \propto \exp\left(-\frac{1}{2}\mathbf{x}'^T\mathbf{\Sigma}^{-1}\mathbf{x}'\right).$$

1. Express $\mathbf{\Sigma}$ in terms of $\mathbf{M}$. Compute it in R from the $\mathbf{M}$ matrix.

2. It turns out that the *covariance* of the data is an estimate for $\mathbf{\Sigma}$. Verify that `cov(Xt)` is indeed close to what you just computed from $\mathbf{M}$.

3. The above should convince you that the matrix $\mathbf{\Sigma}^{-1/2}$ (either the true one or estimated from data using `cov`) will "untransform" the points back to a uniform cloud. Let us test it. To compute the square root of a matrix[5] please use the function below (available in the base code):

   ```
   matrix_sqrt = function(M) {
     e = eigen(M)
     e$vectors %*% sqrt(diag(e$values)) %*% t(e$vectors)
   }
   ```

   Verify that `Xt %*% t(solve(matrix_sqrt(cov(Xt))))` is indeed a uniform point cloud. What we have essentially performed here is also known as *principal components analysis (PCA)*.

Fisher's Linear Discriminant

In the following we shall work with the classic sample *MT Cars* dataset, which is readily available in R as `mtcars`. Use `help(mtcars)` to read more about it. We shall use the `qsec` and `mpg` features as our x_1 and x_2 and the `am` feature as the class label. We drop instances 5, 25, and 32 and normalize the features to zero mean. This is all performed using the following `load.data` function (present in the base code).

```
load.data = function() {
  data = mtcars[-c(5,25,32),]
  x1 = (data$qsec - mean(data$qsec)) # Subtract means
  x2 = (data$mpg - mean(data$mpg))
  y = 2*data$am - 1                  # {0, 1} --> {-1, 1}

  data = list(cbind(x1, x2), y)
  names(data) = c("X", "y")
  class(data) = "data"
  data
}
```

You are also supplied with a helpful `plot.data` method:

```
plot.data = function(data, add=F, cex=3) {
  lbl = (data$y+1)/2  # {-1..1} -> {0..1}
  if (add)
    points(data$X[,1],data$X[,2], bg=lbl, pch=21+lbl, cex=cex)
```

[5]Note that a matrix square root is not uniquely defined, hence you should not expect that $\mathbf{\Sigma}^{1/2} = \mathbf{M}$.

```
  else
    plot(data$X[,1], data$X[,2], bg=lbl, pch=21+lbl, cex=cex, asp=1)
  text(data$X[,1], data$X[,2], col=(1-lbl), cex=0.2*cex)
}
```

Finally, you will need the following function for highlighting points:

```
mark.point = function(x, y=1, bg='red', cex=3) {
  points(x[1], x[2], bg=bg, cex=cex, pch=21+(y+1)/2)
}
```

Exercise 9-11 (3pt). Familiarize yourself with the code. Make sure you can load and plot the data. *Fisher's discriminant*

1. Compute the means of the positive and negative examples. Mark them on the plot using `mark.point`.

   Hint: Use `colMeans`.

2. Let $\mathbf{w} = \mathbf{m}_1 - \mathbf{m}_0$, where $\mathbf{m}_i$ are the class means. Plot the classifier defined by $\mathbf{w}$ (using `plot.classifier`). Add the data points to the plot (use `plot(data, add=T)`). Count (visually) how many points are misclassified.

3. Obviously, the problem is that the data is highly skewed. Compute the covariance matrix of the data `cov(data$X)` and use it to perform covariance normalization as in Exercise 7-8. Visualize the transformed data.

4. Now compute $\mathbf{w} = \mathbf{m}_1 - \mathbf{m}_0$ on the transformed data as before. Visualise and count the number of misclassified examples.

   Hint: To "zoom in" the transformed data on the plot, simply multiply all data by a constant (e.g. `data$X = 3*data$X`).

5. We just used `cov(data$X)` as estimate of $\boldsymbol{\Sigma}$. The original Fisher's discriminant algorithm suggests to estimate $\boldsymbol{\Sigma}$ as an average of two class-conditional covariances:

   ```
   sigma1 = cov(data$X[data$y==1,])
   sigma2 = cov(data$X[data$y==-1,])
   sigma = (sigma1 + sigma2)/2
   ```

   Use this estimate to perform covariance normalization, plot and see whether the new classifier is different.

6. Instead of transforming the data, we can instead transform the weight vector. It turns out that the proper way to compute $\mathbf{w}$ without having to transform the data is simply

   $$\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\mathbf{m}_1 - \mathbf{m}_0).$$

   Compute this vector (on untransformed data and with $\boldsymbol{\Sigma}$ computed as in step 5). Visualize the classifier and the untransformed data.

7. To complete the implementation of the Fisher's discriminant, we have to choose the bias term w_0. The traditional choice is to have the point $\mathbf{p} = 0.5(\mathbf{m}_1 + \mathbf{m}_0)$ lie on the separating line. Find this w_0. Plot the final result. Mark the location of $\mathbf{p}$ using `mark.point` on the plot.

8. Congratulations, you have implemented Fisher's discriminant! Now compare your implementation to a library function:

```
library(MASS)
lda(data$X, data$y)$scaling
```

Least-squares Classifier

Exercise 12 (1pt). Define

Least-squares classifier

```
n1 = sum(data$y == 1)
n0 = sum(data$y == -1)
```

Now define a vector $\mathbf{y}'$ such that:

$$y'_i = \begin{cases} \frac{1}{n_1} & \text{if } y_i = 1 \\ -\frac{1}{n_0} & \text{otherwise.} \end{cases}$$

Finally, compute $\mathbf{w}$ using the least squares rule:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}'.$$

Plot the resulting classifier. Verify that the resulting weight vector is equal, up to a constant, to the Fisher discriminant with $\mathbf{\Sigma}$ estimated as `cov(data$X)`.

If you want to know why and when this happens, meditate at the two equations below for a minute and you shall see:

$$\text{Fisher's discriminant: } \mathbf{w} = (\quad \mathbf{\Sigma} \quad)^{-1}(\mathbf{m}_1 - \mathbf{m}_0)$$
$$\text{Least-squares: } \mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}(\quad \mathbf{X}^T\mathbf{y}' \quad)$$

Perceptron

Exercise 13-14 (2pt). The perceptron algorithm is based on batch or on-line gradient optimization of the error function:

Batch perceptron

$$\mathcal{E}(\mathbf{w}) = \sum_{\mathbf{x}_i \text{ is misclassified}} -\mathbf{w}^T\mathbf{x}_i y_i$$

The expression $\mathbf{w}^T\mathbf{x}_i y_i$ is called the *functional margin* of the training example $\mathbf{x}_i$.

1. Prove that a training example is misclassified *iff* its functional margin is negative.

2. Implement the function `df(w, data)` that computes the gradient of $\mathcal{E}$.

3. Start with `w = c(0, 0)`. Update $\mathbf{w}$ using a single step of the perceptron algorithm: $\mathbf{w} = \mathbf{w} - \mu\,\mathrm{df}(\mathbf{w}, \mathrm{data})$ and plot the classifier with the data (using `plot.classifier` and `plot.data`). Repeat the step and plot again. Repeat iterating and plotting until the algorithm converges to a solution. How many steps did it take?

   Hint: Use `par(mfrow=c(3,3))` to put multiple plots on a single figure.

4. Try changing μ. Does anything change besides the scale?

5. Optional bonus: try using the `animation` package as follows:

```
library(animation)
ani.options(interval=0.2)
ani.start()
# ... Algorithm loop which outputs a number of plots
ani.stop()
```

Exercise 15 (1pt). Implement the on-line perceptron algorithm and visualize its convergence as in the previous exercise. The algorithm selects a single misclassified example on each step. Try highlighting this example on each iteration's plot using `mark.point`.

Online perceptron

Exercise 16 (1pt). In two previous exercises you implemented the perceptron algorithm without the bias term w_0. Modify the algorithms to also search for w_0. Note that you may do it implicitly by simply adding a column of ones to the data matrix. However here I ask you to do it explicitly (in addition, this will integrate nicely with the current plotting logic).

Perceptron with bias

Exercise 17 (1pt). By now you should have implemented the *Fisher's discriminant*, the *Least squares algorithm*, the *Batch perceptron* and *Online perceptron* algorithms, each of them found its own weight vector. Now use *Logistic regression* (the `glm` function, see lecture slides) and *SVM* (the `svm` function from the `e1071` package, see lecture slides). Verify that the weights found by all algorithms are similar up to a constant.

Library functions

Hint: For the `svm` model use `scale=F`. You can then recover the weight vector and bias term as follows:

```
w_svm = t(m$coefs) %*% m$SV
w0_svm = -m$rho
```